

Nonprovisional Patent Application Filed Under 35 U.S.C § 371

Title: **INTERPRETER**

**PRIORITY:**

SWEDISH PATENT APPLICATION NO. 0002440-6, FILED 28 JUNE 2000

Inventors:

FREDRIK LARSSON  
TUNVAGEN 1, 2 TRP  
SE-170 68 SOLNA  
SWEDEN  
A CITIZEN OF SWEDEN

BENGT WERNER  
MOSSVAGEN 1  
SE-194 84 AKERSBERGA  
SWEDEN  
A CITIZEN OF SWEDEN

PETER MAGNUSSON  
BRAXENVAGEN 7A  
SE-181 30 LIDINGO  
SWEDEN  
A CITIZEN OF SWEDEN

**Certificate of Express Mailing**

I hereby certify that this correspondence is being deposited with the U.S. Post Office as Express Mail in an envelope addressed to: Assistant Commissioner for Patents, Box

**PATENT APPLICATION**, Washington, D.C. 20231 on

**JUNE 28, 2001.**

The Express Label No. is **EL922961923US**

*David D. Stei*

Name

**JUNE 28, 2001**

Date

## INTERPRETER

### Technical field

The present invention relates in general to interpreters for executing computer programs realizing, for example, simulators or emulators. More specifically, the present invention relates an apparatus, a method and a computer program product realizing an interpreter providing statistic control encoding for frequency based clustering of service routines in computer program execution.

### 10 Background

With the growing complexity of computer software and the growing reliance on software systems in everyday life and business, there is an increasing need for a high performance in the execution of the software. Performance of software, such as program simulators and emulators, is often crucial since these programs are meant to simulate a system or, like an emulator, to mimic the behavior of a specific system embedded in a larger program complex. The execution of software programs depends on the operation of compilers and interpreters that translates or interprets instruction sets to machine executable code. To achieve high performance of an interactive simulator in cases when instruction sets are not easily decoded, the best practice is to translate the instruction into an internal format, which in this text is called an intermediate format. The intermediate format code is faster to decode and execute for a software-implemented interpreter and can be used directly if the instruction is decoded and executed again.

To achieve high performance, prior art software interpreters usually limit parameter space to an instruction, for example by mapping an instruction to different service routines depending on the parameters given to the instruction. For example, the sparc instruction 'add' may be mapped to two different service routines, one that adds two registers and one that adds a register with a constant. Therefore, the number of service routines will typically be higher than the number of instructions in the simulated instruction set. When the number of service routines grow to a large number, certain compiler optimizations with a high time complexity will make compilation times unacceptable or will fail completely.

### Prior Art

An example of prior art is found in U.S. Patent No. 5,905,895, to Halter, showing a method for optimizing interpretation by carrying out a static analysis of the target or guest

binary code. Pairs or sequences of byte code are replaced by an optimized byte code, thereby in effect optimizing predefined sequences of instructions.

U.S. Patent No. 5,623,617, to Davidian, also optimizes predefined sequences of instructions by recognizing repeated sequences of the same instruction or commonly used groups of instructions.

Another piece of prior art is found in U.S. Patent No. 5,889,996, to Adams, which describes a method to change the instruction cache policy in order to improve the performance of a virtual machine. In practice, this method may be hard to use since there is no access from a user level program to do such changes.

U.S. Patent No. 6,044,220 describes a hardware support using a hash table to aid decoding target or guest instructions

Other pertinent literature describing background art are, for example, Bell, J.R. 1973. "Threaded Code." Communications of the ACM 16, no. 6 (June): 370-372, and Bedichek, R. 1990. "Some Efficient Architecture Simulation Techniques." In USENIX - Winter '90, 53-63

Finally, the JP Abstract No JP 11296379 shows briefly how to optimize a memory arrangement.

#### Object of the Invention

An object of the present invention is to solve the problem of achieving an improved interpreting method and interpreter for computer programs avoiding the drawbacks and problems of prior art interpreters.

An aspect of the problem to be solved is to decrease the overhead in certain service routine structures.

#### Summary of the Invention

The present invention provides an optimization based on dynamically generated execution frequencies that optimize the fetch-decode-execute cycle for frequent individual instructions. Statistics is used to group frequently used service routines in the same program function and to control encoding of instructions. Frequently used service routines are assigned shorter codes thus enhancing the performance of a simulator or emulator and solving problems related to computer compiling speed.

Hence, the present invention provides an interpreter performing operations specified in a computer program consisting of instructions, said instructions being translated to an intermediate format comprising an intermediate code for said instructions, using a service

routine to perform the semantics of an instruction. Embodiments of the interpreter thus comprises:

statistics means for collecting and recording or generating statistics of how often service routines are executed and what parameters they had;

clustering means for grouping frequently used service routines with program jumps between each other in a program function with regard to a predetermined frequency value for determining such service routines;

said statistics means recording the frequency of service routines executed after a said function; and

encoding means for assigning a frequently used service routines a shorter code than service routines executed after a said function, thus, gathered statistics, from an execution, control an encoding to optimize frequently used service routines for faster execution speed.

In one embodiment of the present invention, more than one program function is grouped by grouping frequently used service routines with jumps between each other in the same function, thus minimizing jumps between functions.

Another embodiment provides encoding means for a reduced bandwidth to an electronic memory when fetching intermediate code, as a frequently executed instruction is shorter.

A further embodiment comprises that said statistics is collected before a simulator is compiled.

A still further embodiment comprises that said statistics is collected while a simulator is running, in which case, the simulator dynamically updates a function with service routines used to simulate an instruction set.

Yet another embodiment comprises that said statistics and the encoding provides that a realistic instruction set that is translated to service routines can be written in a high-level programming language for a specific simulator task.

In yet one embodiment the interpreter is provided to be written in the standard ISO C through said statistics and encoding.

A still further embodiment encompasses that a program branch to a next service routine may be shared by all service routines in a function through said statistics and encoding.

A further embodiment reduces the number of jumps, which are hard to predict, in a branch prediction table, causing a table to functioning better on current processor architectures.

One other embodiment sets forth that a profile driven compilation can be used to further enhance a simulator performance through said statistics and encoding.

Yet another embodiment provides for automatic compiler register mapping of often used variables, as the variables may be allocated as local variables.

Another embodiment of the present invention provides for avoiding use of compiler specific extensions, thus providing for compiler independence.

A further embodiment provides for improving instruction cache performance by placing frequent codes in a sequential block due to a common function for service routines.

Another embodiment provides that the interpreter is used by an emulator.

The present invention also sets forth a method for an interpreter, performing operations specified in a computer program consisting of instructions, said instructions being translated to an intermediate format comprising an intermediate code for said instructions, using a service routines to perform the semantics of an instruction. The method comprising the steps of:

collecting and recording statistics of how often service routines are executed and what parameters they had;

grouping frequently used service routines with program jumps between each other in a program function with regard to a predetermined frequency value for determining such service routines;

said statistics recording the frequency of service routines executed after a said function; and

encoding for assigning frequently used service routines a shorter code than service routines executed after a said function, thus gathered statistics, from an execution, control an encoding to optimize frequently used service routines for faster execution speed.

The method is also able to set forth embodiments of the interpreter above as described through attached dependent claims.

The invention is conveniently realized as a computer program product comprising computer program code devised to adapt a data processor or a computer system to perform the steps and functions of the inventive method and apparatus.

### Definitions

This section contains definitions and explanations of terms used in this text.

A code that runs on a host to perform the semantics of a simulated instruction is named a service routine.

A host is a computer the simulator is running on.

An intermediate format is defined as the way to structure and store intermediate code.

Intermediate code is instructions translated into a coding or internal code format that is easy for a software-based interpreter to interpret.

- 5 An interpreter is a hardware or software structure that performs operations specified in a program consisting of instructions.

### Brief Description of the Drawings

- 10 Henceforth reference is had to the attached figures for a better understanding of the present invention and its preferred embodiments, wherein:

Fig 1 schematically illustrates how all service routines are in one function according to prior art;

Fig 2 schematically illustrates only one service routine per function according to prior art;

- 15 Fig 3 schematically illustrates how service routines are grouped into functions with one or more service routines per function according to prior art; and

Fig 4 schematically illustrates frequency clustering according to the present invention.

### Detailed Description of Embodiments

- 20 A central interpreter implemented in a high-level language e.g. "C" can be classified into one of three categories depending on its structure regarding service routines (SR), a code that performs the semantics of a simulated instruction. Either into one of the following categories all service routines in one function see Fig. 1, SR1, SR2, SR3...SR100, only one service routine per function, see Fig. 2, SR1, SR2, SR3...SR100, or the service routines are
- 25 grouped into program functions with one or more service routines per function, see Fig. 3, according to prior art. A program function is marked up by a rectangle in attached Fig. 1 to Fig. 4. An instruction set is a set of operations and their encoding. The operation changes an internal state or performs an input or output. Examples of instruction sets are the Sparc v9 instruction set, which is typically interpreted by hardware, and the java virtual machine
- 30 instruction set, which is typically interpreted by software.

In Fig. 3 the depicted two groups comprise service routines SR1...SR10 and routine SR91...SR100, respectively, other service routines SR11...SR90 are not shown for reasons of simplicity. Each group makes up a program function.

Fig. 1 schematically illustrates how all service routines are in one function according

to prior art. The “all-SR” illustrated in Fig. 1 allows a compiler to accomplish various optimizations that affect several service routines, such as register allocation and code hoisting. Also, it comprises a low overhead when a software jump from one service routine to the next is accomplished. A serious problem is that when the number of service routines grows to a large number, certain compiler optimizations with a high time complexity will make compilation times unacceptable or will simply fail.

Fig. 2 schematically illustrates only one service routine per function according to prior art. The “single-SR” according to Fig. 2 does not allow any global optimization or automatic register allocation and will introduce a higher penalty for jumping from one service routine to the next when executing a software program.

Fig. 3 schematically illustrates how service routines are grouped into functions with one or more service routines per function according to prior art. The “group-SR” according to Fig. 3 is a compromise between two extremes such as single-SR and all-SR. When jumping between service routines in a function, it will have the same benefits as the all-SR but will suffer the overhead when jumping between functions.

The present invention introduces statistics to drastically reduce negative effects of the group-SR structure. Statistics are collected that show how often different service routines are executed and what parameters they had. Furthermore, the statistics also record the frequency of different service routines executed after a specific service routine.

All the service routines that are executed frequently are grouped in one function, thus minimizing the jumps between functions. There may exist more than one such group, thus, it may result in more than one function. This is depicted in Fig. 4, which schematically illustrates frequency clustering according to the present invention with two groups in each function, the groups SR89, SR17...SR6 and SR4, SR34...SR16. This is the case if the statistics show that two or more groups of service routines are very common, but jumps between the groups are very infrequent. For example, the Intel x86 architecture has a 16 and a 32-bit mode in which the processor can operate.

A dotted arrow in Fig. 4 depicts seldom called service routines SR3, SR57, SR94...SR64 and frequently called service routines SR89, SR17...SR6 and SR4, SR34...SR16 are depicted with unbroken arrows.

Both modes may be common but 32 bit instructions and 16 bit instructions seldom follow each other.

The intermediate format is designed so that frequent service routines can be pointed out using a shorter code than the infrequent service routines. In effect the statistics gathered

from execution will control the encoding to optimize common service routines for speed. The encoding will lead to reduced bandwidth to memory when fetching intermediate code, as the frequently executed instructions will be shorter.

Statistics used to group service routines can be collected before a simulator, an  
 5 emulator that also reports about how the system is behaving internally, is compiled or while the simulator is running, in which case the simulator dynamically updates the set of service routines used to simulate the instruction set.

By providing the present invention, realistic instruction sets translated to service  
 routines can be written in a high-level language, compiled and yield a portable effective  
 10 simulator. For example it would allow writing an interpreter in ISO C and make it portable while still having excellent performance. It allows the interpreter to have a large number of service routines without a negative performance impact. Said interpreter comprises, in one embodiment of the present invention:

statistics means for collecting and recording statistics of how often service routines are  
 15 executed and what parameters they had;

clustering means for grouping frequently used service routines with program jumps  
 between each other in a program function with regard to a predetermined frequency value for  
 determining such service routines;

said statistics means recording the frequency of service routines executed after a said  
 20 function; and

encoding means for assigning a frequently used service routines a shorter code than  
 service routines SR3, SR57, SR94...SR64 executed after a said function, thus gathered  
 statistics, from an execution, control an encoding to optimize frequently used service routines  
 for faster execution speed.

A number of optimizations will be possible to apply. Common code might be shared  
 within the function containing the frequent service routines. For example, the service routine  
 epilogue which, among other things, checks for asynchronous events and fetches the  
 intermediate code for the next instruction is identical for all sequential instructions and may  
 be shared. A branch to a next service routine may be shared by all service routines in the  
 30 function. This reduces the number of software jumps that are hard to predict in branch  
 prediction tables, causing them to comply better with current processor architectures. Profile  
 driven compilation may be used to enhance the simulator performance even more.

The present invention provides that manual work such as doing explicit register  
 mapping of often-used variables is avoided. Instead, they may be allocated as local variables

thus letting the compiler do the register allocation. It will also avoid using compiler specific extensions, such as the GNU C && operator which takes the address of a label, making it more compiler independent.

A method used by an interpreter according to the present invention performs the steps

5 of:

collecting and recording statistics of how often service routines are executed and what parameters they had;

grouping frequently used service routines with program jumps between each other in a program function with regard to a predetermined frequency value for determining such

10 service routines;

said statistics recording the frequency of service routines executed after a said function; and

encoding for assigning frequently used service routines a shorter code than service routines executed after a said function, thus gathered statistics, from an execution, control an

15 encoding to optimize frequently used service routines for faster execution speed. Also, it is able to set forth embodiments of the interpreter above as described through attached dependent claims.

The present invention improves instruction cache performance by placing frequent code in a sequential block as it is placed in a common function.

20 It is appreciated that the present invention can be realized by means of hardware, software or a combination thereof.

The present invention has been described by way of exemplifying embodiments, and it should be appreciated by the person skilled in the art that the invention can be realized in a variety of ways within the spirit of the claims.